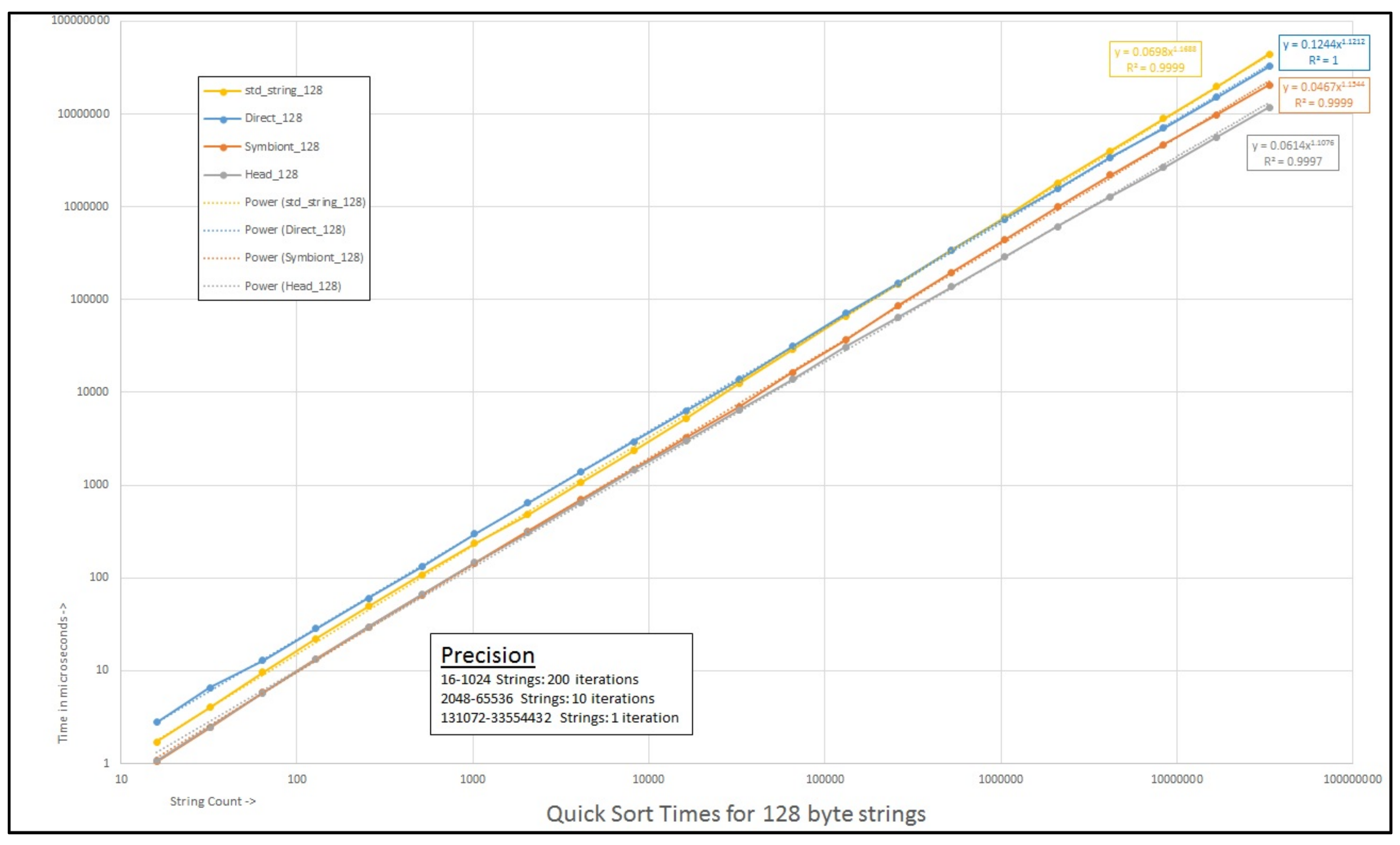
What We Need is a Green C++ Standard Library and a Green C++ Programming Model to Go with It.

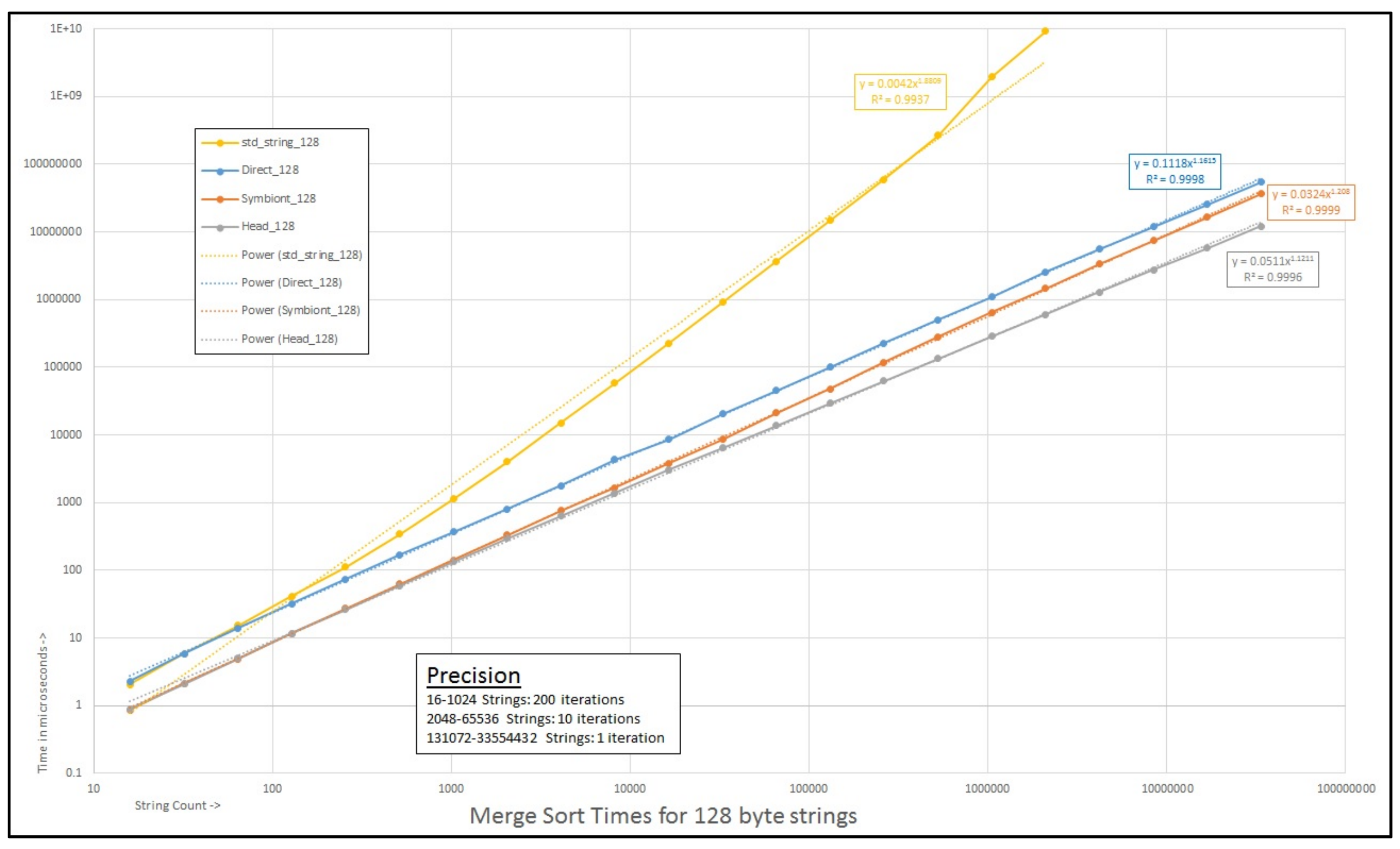
C. Johnson, email: cjohnson@member.fsf.org

GitHub: https://github.com/charlesone/Kaldanes



C++ Standard strings quick sort almost an order of magnitude slower than one of the three new string classes at scale, and this is the best case, the worst case to the right

All execution numbers presented are single-threaded and built for release code, running on a small format HP (Intel 4-core i5-3570) box with 15GB of DRAM running Centos-7 Linux, with the gcc/g++ 4.8.5 C++11 compiler and runtime.

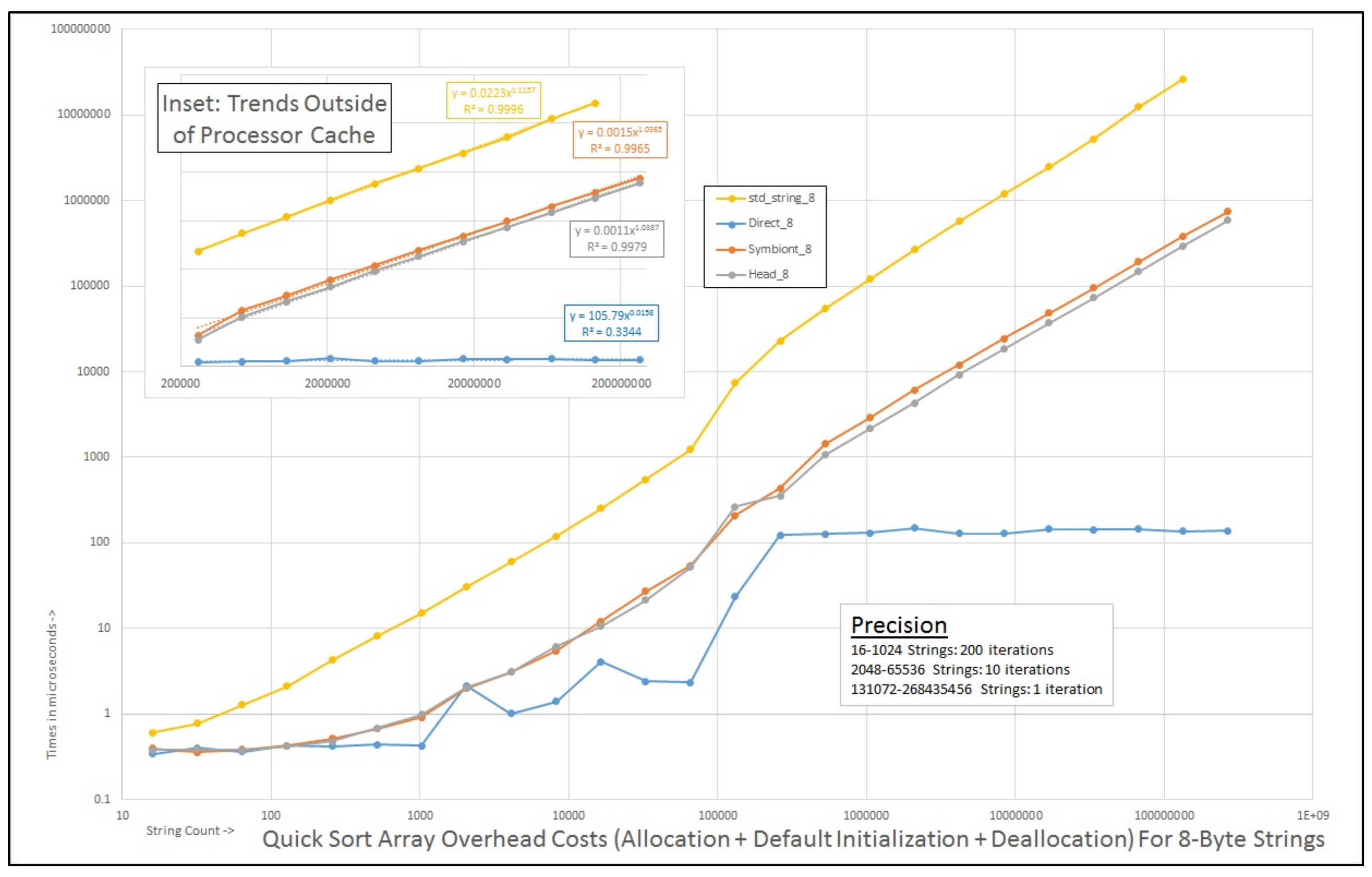


C++ Standard strings merge sort many orders of magnitude slower than all the new string classes at scale, since merge sort uses temporary arrays and std:strings have costly default initialization

The first half of what's needed is much faster and more importantly more scalable C++ classes and containers. Looking at the first two slides, we can start to graphically see what is wrong: replacing the complex C++ types and containers with simpler ones and dispensing with fine grained allocation, initialization and deallocation makes the code execute and scale remarkably well. Another boost comes from using a logical AND physical data format that is base + offset addressed to be allocated in slabs on the stack or mmapped in a DFS file system or Gen-Z librarian file system, versus the slow execution and abominable scalability of standard library types and containers as they are serialized into and back out of messages, buffers and storage media. In a detailed discussion, some of the reasons why are still a standard library and C++ runtime mystery:

https://stackoverflow.com/questions/54562770/performance-of-big-slabs-due-to-allocation-initialization-and-deallocation-over

The fact is, that at the petabyte scale, it takes 302 days of computer time to allocate, default initialize and deallocate the standard library data structures, ***before adding any data to them***, versus 100 microseconds per 10 GB table (which comes to ten seconds per petabyte) using greener data structures, containers and programming models.



Thus, greener C++ overcomes the physical barriers to good C++ performance, to get C++ up to the level of typical C programming performance. This alone would require an entirely new C++ standard library to be written to stop wasting energy in the world's data centers, looming ominously to consume 20% of the world's power in the near future (see Pinar Tozun's HPTS 2015 talk.)

The second half of what's needed is a green C++ programming model dramatically surpassing the C program performance, based on variadic template parameter pack recursion, and generic programming in general, with C++11 features and something new that is dubbed "inference programming" or "inference computing," which executes in the compiler and not at runtime and involves compiling first the code and then the database into "memoized joined row objects" that are shrunken by a factor of 250 for 1000-byte row tables, using both type and data inference. What's left after C++ is done with compiling the code are heavily optimized pointer fetches and heavily optimized data movement, using "poor man's normalized keys" (Goetz Graefe) in the indexing.

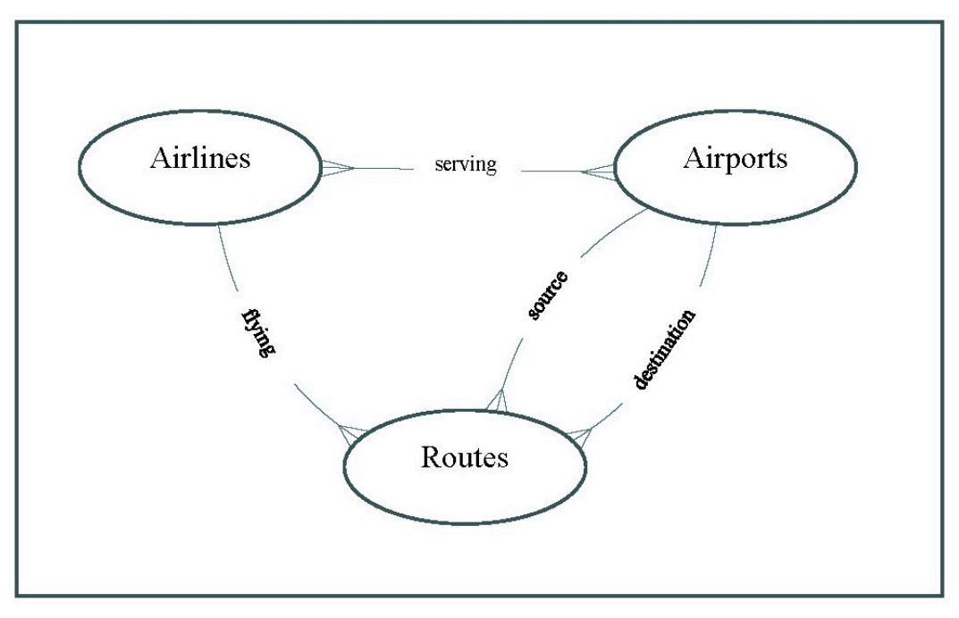
As is stated in the bible for grad students: ***'Cracking the Coding Interview***,' Gayle Laakmann McDowell, 6th Edition., chapter 14 'Databases,' p. 172:

"Large Database Design: When designing a large scalable database, database joins (which are required in the above examples) are generally very slow, Thus, you must denormalize your data. Think carefully about how your data will be used - you'll probably need to duplicate data in multiple tables."

However, we can ignore that advice with enormous shared memory-centric systems like Gen-Z and Intel 3D XPoint and make relational database joins free, because any number of tables can be joined of any size accessing shrunken memoized joined row data to support queries executing in a microsecond or so in the data center, or remotely at the speed of the network:

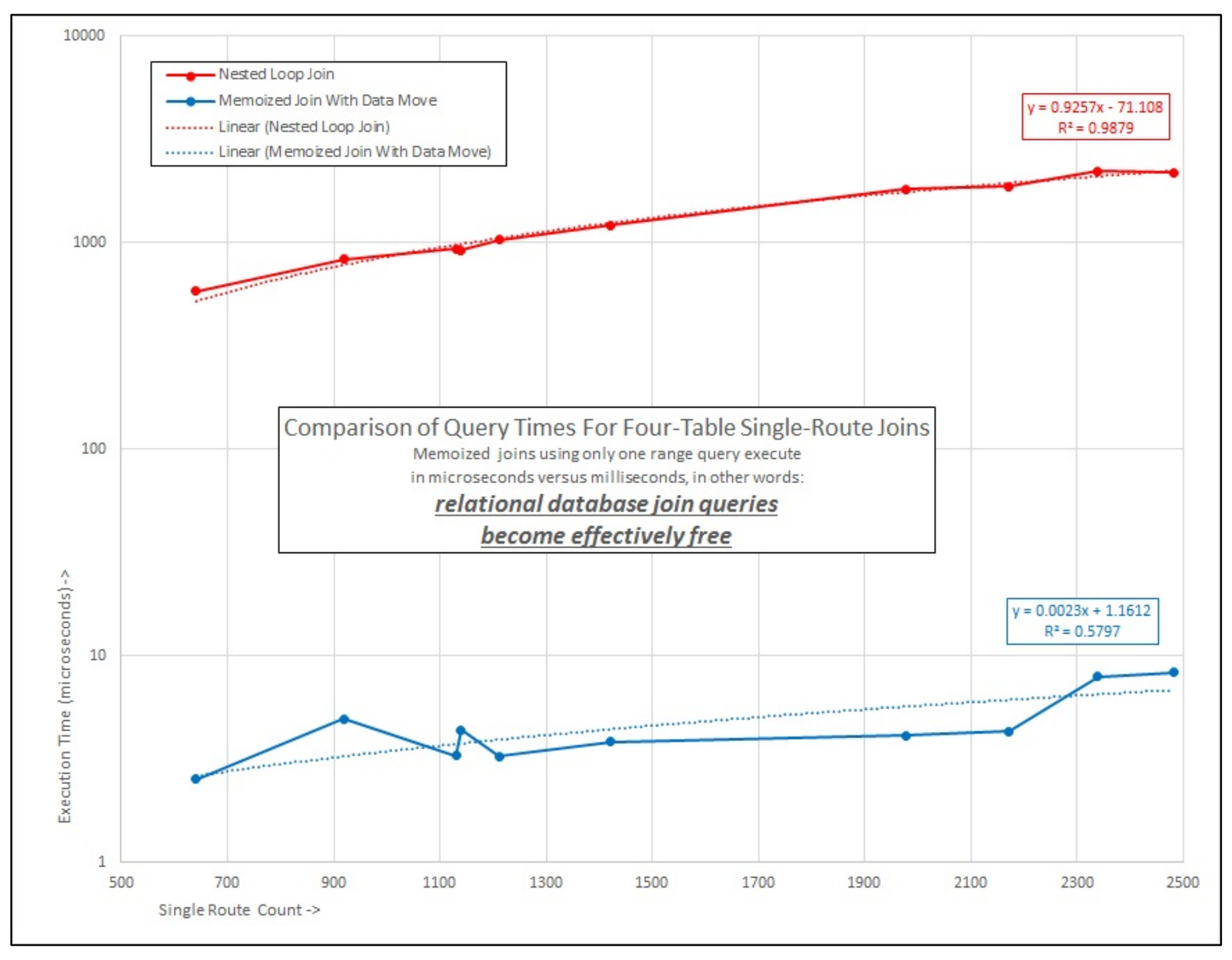
1. New faster string class objects scale better: Direct, Symbiont[+PMNK – stands for "Poor Man's Normalized Key" – Goetz Graefe https://doi.org/10.1145/1140402.1140409], and Head[+PMNK] instead of std::string.
2. For the relational database system, Direct becomes the basis for RowString (with columns) and Head becomes the basis for IndexString (friend class of RowString using PMNK).
2. Slab allocation instead of fine-grained allocators: using automatic variables on the stack, or mmap.
3. Contiguous built-in arrays instead of standard library containers.
4. Relation vectors describing from and to relations between columns of tables, in C++11 variadic template parameter packs as arguments to create joins.
5. Generic programming using recursion across those parameter packs at compile time, with constexpr initJoin and join routines leaving a runtime consisting of pointer fetches and some data movement.
6. Inference programming with the typical SQL DDL, DML and metadata optimized out of the C++11 runtime using const static globals and constexpr operations.
7. Finally, memoized joins replacing materialized views with 4% (for 100-byte rows) or less of the space complexity, but retaining full expressive access by inference using <tuple>.

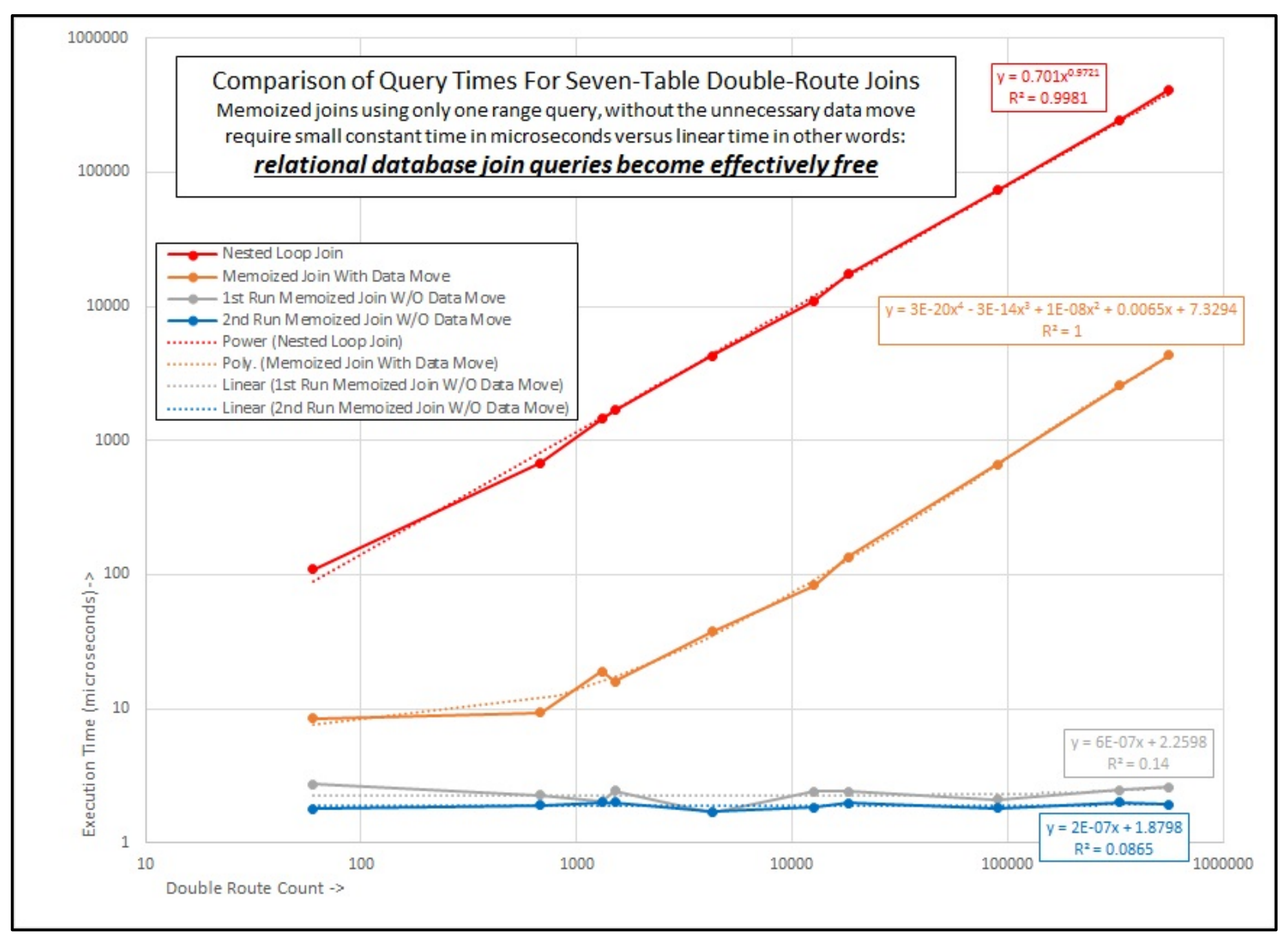
The demo programs use the airport, airline and route data for the world's airlines: (1) **TableDemo** (walks the user through the evolution of the classes), (2) **JoinDemo** (showing nested loop 4-table join times that are less than 10 microseconds: where the program loads 3 tables from CSV files, builds ten indexes and performs 10 4-table joins in less than 1/8 of a second), and (3) **BigJoinDemo** (showing 7-table memoized join times that are flat at 1.8 microseconds on a slow box) are there to show the function, and in debug build can show this performance is not sleight of hand.

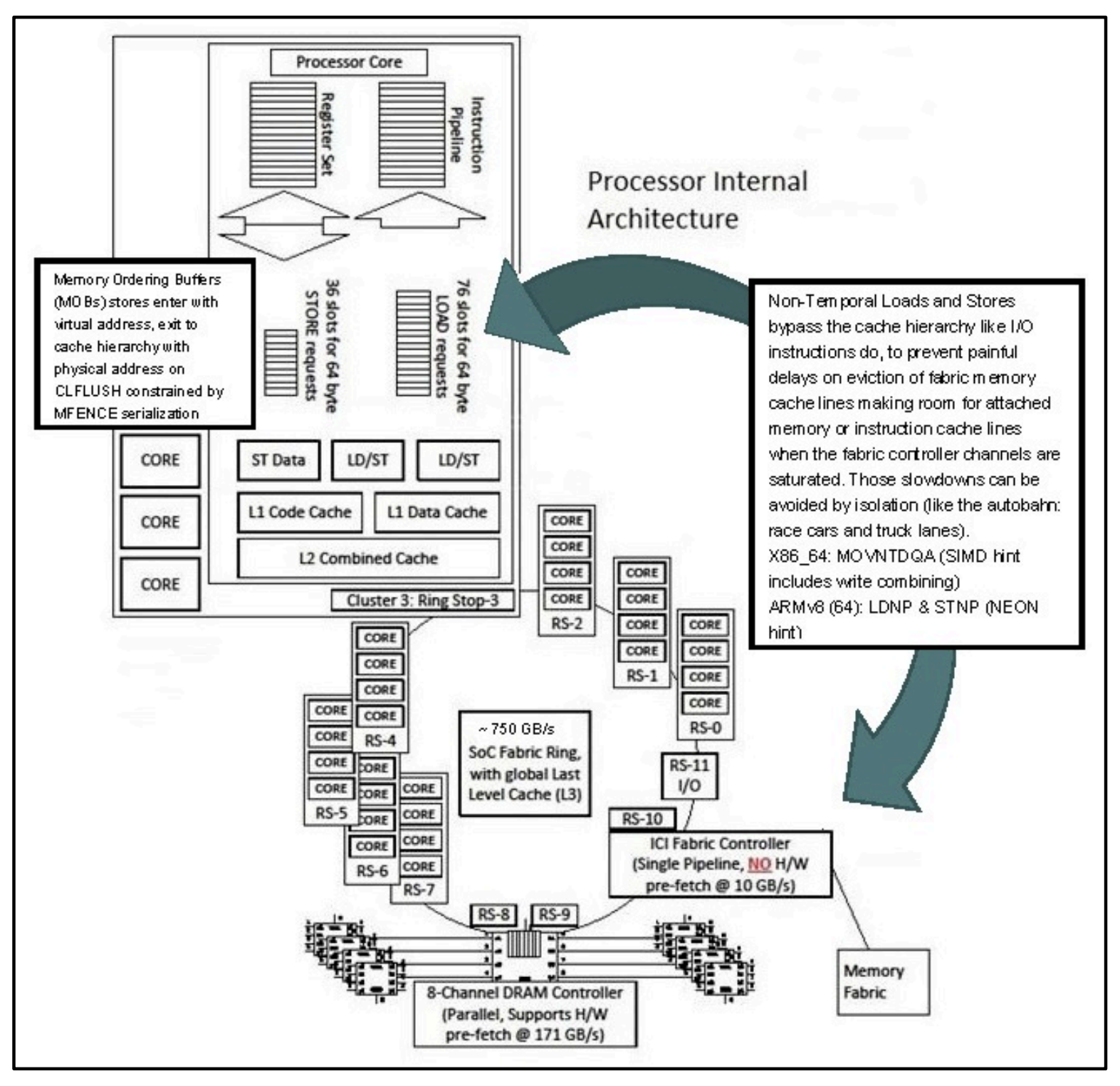


At a high level, there are a collection of variadic template classes: (1) a RowString class to hold table data and (2) its friend class IndexString to index the columns in the rows, (3) a RelationVector class structures the joins in ordered sets of directed column pairs called parameter packs in C++, (4) a heavyweight QueryPlan class to hold the relational database metadata and optimize and construct the joins at compile time using the type system in a constexpr function called initJoin(), and (5) its extremely lightweight friend class JoinedRow which retrieves the query output from the QueryPlan join() constexpr function.

Future designs are discussed in the Technical White Paper on the GitHub: (1) BASE + ACID database systems where these fast BASE relational database joins can be built on-the-fly using snapshots as the ACID database is doing ingest, (2) how to mutate the immutable relational database into a full ACID database support ultra-fast massive join queries without slowing ingest speeds, and (3) how to execute everything in parallel, scaling up AND out.







Green isn't just good for the planet, it transforms relational database joins (and other algorithms) at scale into a real-time proposition.